Заметки по книге Брюса Эккеля «Философия Java» 4-е полное издание.

Главы 1-6

Уборка мусора никак не может быть фоновым процессом, т. к. программа приостанавливается на время работы сборщика мусора.

Как работает сборщик мусора

Схемы сборки мусора

1. Подсчет ссылок.
Медленно. С каждым объектом хранится счетчик ссылок на него. Каждый раз при присоединении ссылки к объекту счетчик увеличивается. При выходе их области видимости переменной или присвоении ссылки значения null счетчик уменьшается. Подсчет создает постоянные задержки во время работы программы. Если объекты создают циклические ссылки друг на друга, счетчик не сможет обнулиться и объекты останутся в памяти.
2. Цепочка ссылок.
Любой объект прослеживается до ссылки в стеке или статической памяти. Для каждой ссылки берется объект, на который она указывает и отслеживаются все ссылки этого объекта. И так далее, пока не будет проверена вся инфраструктура ссылок. Проблемы циклических ссылок здесь не существует, т. к. такие ссылки не обнаружаются и автоматически будут удалены сборщиком мусора.
3. Остановить и копировать.
Используются 2 кучи. Во время работы сборщика мусора живые ссылки копируются из одной кучи в другую. Мусор остается в первой. Сами ссылки при этом меняются, так как поменялось их расположение в памяти. Недостаток в использовании 2 куч и «перелопачивании» памяти то туда, то сюда. При этом половина выделенной памяти тратится впустую

4. Пометить и убрать
Такая же схема, как и предыдущая, только при нахождении мусора ссылка не удаляется, а помечается флагом. Уборка мусора при этом происходит только после процесса пометки ссылок. Если куча фрагментирована, сборщик мусора исправит это путем перемещения объектов внутри нее.

JVM следит за успешностью работы каждой из схем. Когда работа какой-либо из них становится непроизводительной (например: при схеме «пометить и убрать» куча может стать излишне фрагментированной и тогда сборщик мусора переключится к схеме «остановиться и копировать») сборщик мусора переходит у другой схеме. Это и есть *адаптивный механизм* работы сборщика мусора.

Эти механизмы были созданы для освобождения памяти и ускорения работы программы.

Ускорение работы в JVM

Другой способ ускорения программы называется компиляцией «на лету» (Just-In-Time, JIT). Компилятор JIT частично или полностью конвертирует программу в машинный код. Этот код не нуждается в обработке виртуальной машиной и может выполняться намного быстрее. При загрузке класса система находит файл .class и байт-код этого класса переносится в память. В этот момент можно провести компиляцию JIT для кода класса, но: 1. это займет время; 2. увеличивается размер исполняемого файла, что может привести к подкачке памяти и замедлить программу.
Код JIT компилируется только тогда, когда это становится необходимо. Невыполняющийся код никогда не будет компилироваться JIT.

Технология Java HotSpot, встроенная в JDK делает это похожим образом, последовательно оптимизируя код при каждом его выполнении. Т.е. чем чаще выполнятся код, тем быстрее он работает.

Статические поля инициализируются только при инициализации объекта *или* обращении к этому полю через имя класса.
Статический блок выполнятся только при создании объекта *или* обращении к какому-либу статическому полю этого класса.
Порядок инициализации:
1.Статические поля.
2.Статический блок
3.Все родительские конструкторы, начиная с самого старшего
4.non-static поля
5.non-static блок
6.Конструктор класса

Integer и int.
Integer – это объектный тип, int – это примитивный тип.

Integer – это объектная оболочка над int.

Использую перегрузку методов, метод принимает аргумент Integer, а подаю ему int.
Может ли возникнуть проблема?

Если Вы используете перегрузку методов и, например, метод doIt(...) имеет две сигнатуры:

public void doIt(int i) {...}
и
public void doIt(Integer i) {…}

то при вызове doIt(5) будет вызван первый метод, а при doIt(new Integer(5)) – второй.

Если же у Вас есть только метод

public void doIt(Integer i) {...} -

то при вызове doIt(5) значение 5 автоматически упакуется (autoboxing) в Integer.

```
int a = 1000; // a - число
Integer b = 1000; // b - ссылка на объект
Integer a1 = 127;
Integer a2 = 127;
Integer b1 = 500;
Integer b2 = 500;
System.out.println(a1==a2);
System.out.println(b1==b2);
```

Результатом выполнения будет:

```
true
false
```

Здесь фактически вызывается статичный метод java.lang.Integer.valueOf(int), который кэширует значения от -128 до 127:

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}

private static class IntegerCache {
    static final int low = -128;
    static final int high;
static {
    // high value may be configured by property
    int h = 127;
    String integerCacheHighPropValue =
        sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
    if (integerCacheHighPropValue != null) {
        try {
            int i = parseInt(integerCacheHighPropValue);
            i = Math.max(i, 127);
            // Maximum array size is Integer.MAX_VALUE
            h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
        } catch( NumberFormatException nfe) {
            // If the property cannot be parsed into an int, ignore it.
        }
    }…
// range [-128, 127] must be interned (JLS7 5.1.7)
```

Списки аргументов переменной длины

```
public void add(Integer...vals){
    for(Integer v : vals){
        add(v);
    }
}
```

Вызовы:

```
add(45, 10, 90, 80, 91, 54);
Integer[] ints = {1, 150, 45, 87};
add(ints);
add(800);
add(); //Ниичего не будет выполнено
       //(в данном случае, т. к. в цикле нет итераций)
```

Метод с сигнатурой `public void add(Integer []vals)` уже создавать нельзя.
Если добавить такой код:

```
public void add(Long...vals){
    for(Long v : vals){
        add(v);
    }
}
```

То программа не скомпилируется, т. к. непонятно, какой метод вызывать при `add();`
Списки аргументов переменной длины усложняют перегрузку, хотя по-началу выглядят безопасно. Как правило используется только в одной версии перегруженного метода или не используется вообще.

Перечисления

Перечисления являются классами и могут обладать своими методами.

```
enum Spiceness{
    NOT, MILD, MEDIUM, HOT, FLAMING
}

Spiceness spiceness = Spiceness.MEDIUM;
System.out.println(spiceness);
for (Spiceness sp : spiceness.values()){
    System.out.println(sp.name() + " " + sp.ordinal());
}
```

`Ordinal()` обозначающий порядок объявления констант и статический метод `values()`, возвращающий массив констант enum в порядке их объявления, компилятор создает автоматически. Вывод кода выше:

```
MEDIUM
NOT 0
MILD 1
MEDIUM 2
HOT 3
FLAMING 4
```

Особенно enum удобен в командах *switch*:

```
switch (spiceness){
    case FLAMING:
        System.out.println("flaming");break;
    case NOT:
        System.out.println("not");break;
    case MEDIUM:
        System.out.println("medium");break;
    default:
            System.out.println("Another one"); break;
}
```

Пакеты

Пакет = папка. Классы могут иметь одинаковые имена, если находятся в разных пакетах. Чтобы использовать эти классы необходимо воспользоваться ключевым словом import в начале файла. Пример:

```
import nbrb.DailyExRates;
```

Таким образом Java понимает, что необходимо использовать класс `DailyExRates` из пакета *nbrb.*

Если необходимо использовать класс с таким же именем из другого пакета, необходимо при его объявлении указать полное имя класса с пакетом. Пример:

```
bank.DailyExRates dailyExRates = new bank.DailyExRates();
```

В качестве имени пакета лучше использовать свое собственное доменное имя, написанное в обратном порядке. Пример:

```
import ru.razilov_code.MyCollection;
```

Либо любое другое уникальное имя пакета (особенно, если планируется выкладывать написанные программы в интернет или кому-то отдавать).

Так как в Java программе может использоваться большое количество классов, необходимо их упорядочить по папкам (пакетам). Имя пакета указывается в начале файла до инструкций *import*:

```
package ru.razilov_code;
```

В Java имя *public* класса всегда должно совпадать с именем файла без расширения. В одном файле может быть только один *public* класс.

В файле может вообще не быть *public* классов и тогда файл может иметь любое имя. Однако это будет вызывать трудности при чтении кода.

JVM работает по схеме:
Проверяет переменную окружения CLASSPATH (ее значение задается операционной системой, а иногда программой установки Java или инструментарием Java) . CLASSPATH содержит список одного или нескольких каталогов, используемых в качестве корневых при поиске файлов .class. Начиная с этих корневых каталогов JVM берет имя пакета и заменяет точки на слэши. Т.е. имя пакета ***ru.razilov_code*** преобразуется в путь ***ru/razilov_code***. Затем это имя присоединяется к элементам CLASSPATH. В полученных местах ведется поиск файлов .class, имена которых совпадают с именем создаваемого программой класса. (Поиск так же ведется в стандартных каталогах, определяемых местонахождением интерпретатора Java). Т.е., если полное имя класса *ru.razilov_code.MyCollection*, а переменная CLASSPATH имеет вид `CLASSPATH = .; C:\Admin\Java;`, то JVM будет этот класс искать по пути `C:\Admin\Java\ru\razilov_code\MyCollection.class`, а так же в текущем каталоге (символ точка).

Для файлов JAR немного иначе. В переменной CLASSPATH должен храниться полный путь к файлу JAR.

Конфликты имен
Предположим, у нас есть 2 класса *MyCollection* в разных пакетах:

```
import ru.razilov_code.*;
import ru.razilov_code.collectoins.*;
```

Ошибки не возникнет до тех пор, пока мы не начнем объявлять объект класса *MyCollection:*

```
MyCollection mc = new MyCollection(); //Выдаст ошибку
ru.razilov_code.MyCollection mc = new ru.razilov_code.MyCollection(); //Правильно
```

Использование импортирования для изменения поведения

Использование одинаковых имен классов можно использовать для отладки.
Например: у нас есть 2 пакета *debug* и *debugoff*. В каждом из них есть класс *MyCollection*.
Методы этих классов имеют одинаковую реализацию, за исключением некоторых моментов. В первом пакете этот класс имеет такой метод:

```
public static void doSomething(){
    Random r = new Random(47);
    int [] arr = new int[r.nextInt(50)];
}
```

Во втором пакете этот метод имеет следующи вид:

```
public static void doSomething(){
    Random r = new Random(47);
    int n = r.nextInt(50);
    int [] arr = new int[n];
}
```

Такой код отлаживать проще. Кроме того, методы могут иметь совершенно разную реализацию. Таким образом изменив имя пакета в *import* можно изменить полностью поведение класса.

Чтобы импортировать статические члены класса используется команда

```
import static razilov_code.MyCollection.stClass.*;
```

Вместо * может быть имя статического метода или статического класса.

Спецификаторы доступа в Java
public — члены класс доступны всем,
private — члены класса доступны только внутри класса,
protected — члены класса доступны внутри пакета и в наследниках;
Без модификатора — члены класса видны внутри пакета .
Если в начале файла не указано имя пакета, то будет использован «пакет по умолчанию» для текущего каталога (Для этого переменная CLASSPATH должна содержать символ точки!)

При пакетном доступе класс-потомок наследует метод/поле без модификатора ТОЛЬКО если класс родитель объявлен в том же пакете.
Если класс-потомок создать в другом пакете, то пакетные члены не наследуются.

При уровне доступа protected класс-потомок наследует protected члены, находясь в любом пакете.

Чтобы выполнить метод, находящийся во внутреннем классе, можно использовать такую конструкцию:

```
new NiksSon2().new Cl().get()
```

Выполняет метод *public get()* из внутреннего *public* класса *Cl*.
Классы могут иметь только модификаторы *public* или *friendly* (пакетный доступ).
(Внутренние классы могут иметь модификаторы private и protected, но это другая история)

Чтобы ограничить доступ к конструктору класса, надо создать приватный конструктор, а объекты возвращать в статическом методе. Это может понадобиться, например, для ограничения количества создаваемых объектов. Пример:

```
class NewClass {

    private int i;
    private static NewClass newClass = new NewClass(50);

    public static NewClass getInstance(){
        return newClass;
    }

    private NewClass(int i) {
        this.i = i;
    }

    public int getI() {
        return i;
    }
}
```

Нет возможности создать класс через конструктор из другого класса. Можно только получить единственно возможный экземпляр класса с помощью статического метода *getInstance()*.

Чтобы ограничить количество создаваемых объектов, например, до 20 можно воспользоваться следующей конструкцией:

```
package Jsonvaluta;

public class ConnectionManager {
    private static int cnt;

    public static int getCountOfCreatedConnections() {
        return cnt;
    }

    private ConnectionManager(){}

    public static Connection getNewConnection(){
        if(cnt == 20) return null;
        else {
            cnt++;
            return new Connection();
        }
    }

    @Override
    protected void finalize() throws Throwable {
        cnt--;
    }
}
```

```
package Jsonvaluta;

public class Connection{
    Connection(){
    }
}
```

Существует 2 причины для ограничения доступа к членам класса:
1. Предотвращение использования клиентами внутренней реализации класса, не входящей во внешний интерфейс. Объявление полей и методов с модификатором private помогает пользователям класса сразу видеть, какие члены класса для них важны, а какие можно игнорировать. Это упрощает понимание и использование класса.

2. Возможность изменения внутренней реализации класса, не затрагивающего программистов-клиентов. Пример: сначала вы реализуете класс одним способом, а затем понимаете, что можете повысить скорость его работы. Отделение интерфейса от реализации позволит сделать это без нарушения работоспособности существующего пользовательского кода, в котором этот класс используется.

Открытый интерфейс класса — это то, как видит его пользователь. Даже если идеальный интерфейс не удалось построить с первого раза, можно *добавить* в него новые методы без удаления уже существующих, которые могут использоваться программистами-клиентами.